NG Tools Documentation

Author	Michaël Nguyen
Package Version	0.8.30
Date	3 February 2017
Online Documentation	Documentation
Bug Tracker	Support
Contact	support@ngtool.tech

Table of Contents

The package was tested with:

- 5.6.0b3
- 5.5.0f3
- 5.4.0f3
- 5.3.0f4
- 5.2.0f3
- 5.1.0p3
- 5.0.0f4
- 4.7.1f1
- 4.6.1f1
- 4.5.0f6

NG Tools is a gathering of many useful tools to improve your experience through **Unity**. It embed many user-friendly features that will change your way to do things, your efficiency will be undoubtful, once your get in there, you will never go back, be aware, you are going to be addicted!

1 Usage

1.1 NG Console

NG Console is like **Unity**'s **Console**, but way more efficient. Based on modules, each module brings a feature that might be useful to you. Below is the explanation of each built-in modules.

Base features

All features from the native **Console** (**NG Console** is strongly linked to the **Console**, all primary features are sync'ed):

- **Intercept logs from Unity Console** Any logs entering **Unity** console are catched and displayed in **NG Console** as well.
- **Clear** Clear the console.
- **Collapse** Logs are stacked, reducing number of visible logs. (Note that this option is only available in the main **Stream log**. See after.)
- **Break on error** Break the game when an error occurred.
- **Clear on play** Clear the console whenever the user starts to play.
- **Persistent logs** Logs are saved during the session of **Unity** (Saved between serializer's passes).
- **All types log** All types can be toggled (Normal, Warning, Error).
- **Auto-scroll** Stick the scrollbar to the bottom when new logs come if the scrollbar was already stick to the bottom.

Additional features

- **Type log exception** Error is split between Error and Exception. They are totally dissociated.

- **Interactive stack trace** Display all frames from the stack trace. Each frame is clickable if the file is editable.
    - **Colored source code preview** When the mouse hovers a stack frame, if the file is reachable in your hard disk, the source code is displayed in colors.
    - **Folder pingable** Directly ping folder by clicking on it in the stack frame.
    - **Discard frame** You can choose to discard particular namespace, class or function from the stack frames. Use it to filter out your custom logger.
- **Advanced copy** Copy the first line of the selected logs or its full content or the whole stack trace.
- **Log icon** An icon representing the emitter is drawn just after the log's foldout if a context is provided. Clicking on it will ping the object.
- **Time** Logs can display the time, but only when it was received by **NG Console**, therefore a little delay might be detected. (Thus, the time is not reliable!)
- **Drag** If you provide a context to the logger, you can drag it from the log in **NG Console**.
- **Multi selection** You can select many logs.
- **Multi copy** You can copy from all selected logs, it will append each log in the clipboard.
- **Multi deletion** You can temporary delete selected logs, unfortunately it is not persistent between **Unity** serializer's passes.
- **Modular context menu** Context menu on log is modular. Any module or filter can implement a its menu item extension.
- **Export logs** Export all logs from a stream in your desired format. There are many options using the export wizard, giving all keys to export to your own format.

1.1.1 Module

A module implements any behaviour you might will. Easy to implement new features or to interact with others.

- **Main module** Mimics the native **Unity** console. It displays incoming logs through streams. It separates normal logs and compiler's in different stream.
    - **Stream log** You can create many streams to display logs. Each streams can be customized to display specific logs depending on filters.
        - **Log**
            - A log is the content you write to the console. But there are many ways to display a log. Beside the default log just containing a message and a context, **NG Console** provides extra functions to output logs in different ways.
            - **See**: How to implement a Log (called as Row in the engine).
            - **See**: Additional loggers.
        - **Filter**
            - A filter accepts or rejects incoming logs regarding its condition.
            - **ContentFilter** accepts logs by matching a keyword in it.

        - **HierarchyFilter** accepts logs coming from a *GameObject* or its children.
        - **MaskTypeFilter** accepts logs matching a type (Normal, Warning, Error or Exception).
        - **NameHierarchyFilter** accepts logs by matching a keyword with log's Object's name.
        - **TagFilter** accepts logs coming from *GameObject* with a certain tag.
        - **See**: How to implement a filter.
- **Archive module** Logs can be archived, each archived log can be assigned a note and put in folders.
    - **Folder** You can create many folders and name them. Drag & drop archived logs between folders to keep a neat archive place.
- **Color Markers module** Colors log's background regarding markers' filters and color. You can also add temporary background color. These disappear after compilation.
- **Remote module (Command Line Interface)** Implemented especially for **NG CLI**, allowing **NG Console** to be able to receive logs from a remote **Unity** build. Also, if the distant build has activate CLI, you will be able to remotely execute commands on the build.

See: How to implement a Module for **NG Console**.

1.1.2 Editor openers (Only for those who got problems with Unity console's `Go to line`)

NG Console can natively open any of the following editors with the `Go to line` feature:

- VisualStudio
- MonoDevelop
- Sublime 2
- Notepad++
- Vim
- Emacs

If your editor is missing, you can technically implement it yourself, since **NG Console** uses the same technic as **Unity**, otherwise just ask for it, I will be pleased to give you assistance!

Some numbers

- **CPU Footprint** Can handle **1M** and certainly even more, way more!
- **Memory Footprint** With all the features brought by **NG Console**, it only takes 1.5 times the memory of **Unity** console.

See: Look at **NG Console**'s settings in **NG Settings**.

1.1.3 Loggers

NG Console provides the debug class *NGDebug* containing many useful loggers.

```
NGDebug.LogHierarchy(GameObject gameObject)
NGDebug.LogHierarchy(Component component)
NGDebug.LogHierarchy(RaycastHit hit)
```

Writes the hierarchy of the given *Object*, as a list of pingable button for each parent.

```
NGDebug.Log(params Object[] objects)
NGDebug.Log(params RaycastHit[] hits)
NGDebug.LogCollection(IEnumerable<RaycastHit> hits)
NGDebug.LogCollection<T>(IEnumerable<T> objects)
```

Writes a log containing a pingable button for each reachable object of the given list. Use it with GetComponents() to instantly have a visible list of all components in **NG Console**.

```
NGDebug.MTLog(string message)
NGDebug.MTLog(string message, Object context)
```

Writes a log containing the stack trace. Useful when writing logs in multi-threads context.

```
NGDebug.Snapshot(object o)
NGDebug.Snapshot(object o, Object context)
NGDebug.Snapshot(object o, BindingFlags bindingFlags)
NGDebug.Snapshot(object o, BindingFlags bindingFlags, Object context)
```

Output all fields and properties matching the BindingFlags from the given object to the console.

1.2 NG Game Console

1.2.1 NG Game Console

Runs a console window in-game displaying incoming logs and additional data. Data might be anything you want to display about your scene, your environment, your game, your hardware, your metrics, etc...

There is 2 rendering modes, *Logs* and *Data*:

- *Logs* displays logs and has a header filled with short data version. *Logs* possesses 2 terminal modes:
  - *Raw Text* displays logs as plain text.
  - *Logs* displays logs as real separated entities.
- *Data* displays a list of all data in their complete version.

How To Use: Drop the script *NG Game Console* on a *GameObject*. If you want to add data to the console, add any script inheriting from *DataConsole* on a *GameObject* and put it in *NG Game Console*'s field "Data Console".

See: How to implement a DataConsole for **NG Game Console**.

1.2.2 NG CLI

NG CLI allows to type and execute commands in the game.

There is 2 ways to use it:

- By linking it to a **NG Game Console**, **NG CLI** provides an input at the bottom of the console to type commands.
- Through an instance of **NG Server Command**, it allows you to execute commands via network from the module **Remote** in **NG Console**.

Commands are exposed through **NG CLI**'s field "Root Commands". Fill this field with any *Behaviour*. Every properties (with both Get and Set) and methods with the attribute *CommandAttribute* will be exposed as sub-commands of the *Behaviour*.

How To Use: Drop the script *NG CLI* on a *GameObject*. To make it appear into a console, you need to assign the field "Game Console" with a script *NG Game Console*.

How To Use: To use it via network, drop the script *NG Server Command* on a *GameObject* and assign the field "Cli" with a script *NG CLI*.

See: How to implement a root command for **NG CLI**.

See: Refer to the class *CommandAttribute* for more informations.

See: Look for network guidances if you encounter issues.

1.2.3 NG Server Command

NG Server Command grants the possibility to send debug logs to module **Remote** in **NG Console**. If field *Cli* is assigned, module **Remote** is allowed to remotely type and execute commands.

How To Use: Drop the script *NG Server Command* on a *GameObject*, then drop any script inheriting from *AbstractTcpListener*, this script is required to tell the server how to communicate with the outside.

Note: *DefaultTcpListener* is the default built-in listener.

See: How to implement an AbstractTcpListener for a server.

See: Scene GameConsoleCLI to see an example of **NG Game Console**, **NG CLI** and **NG Server Command**.

1.3 NG Remote Scene

A powerful tool to modify the scene on your device, but from **Unity Editor** itself!

With **NG Remote Scene** you can easily test many configurations on your build and rapidly iterate without the need to rebuild, copy, launch, test.

NG Remote Scene was built to work from an empty project on any device with **NG Remote Scene** activated. Meaning you do not need to have the project or any script to have it working.

How To Use: Drop the script *NG Server Scene* on a *GameObject*, then drop any script inheriting from *AbstractTcpListener*, this script is required to tell the server how to communicate with the outside.

Note: *DefaultTcpListener* is the default built-in listener.

See: Look for network guidances if you encounter issues.

1.3.1 NG Remote Hierarchy

Equivalent of **Hierarchy** window in **Unity Editor**. This is the base window from where you connect to the device, change settings, change global behaviours of **NG Remote Scene**.

It supports base features of *Hierarchy* window:

- **List full hierarchy**
- **Drag & Drop**
- **Filtering**
- **Delete GameObject**

Options

- **Network Refresh**: An interval in second between each send of packets.
- **Sync Tags**: Defines if you choose to use the tags in your current project or let an input text.

Note: You can technically have many **NG Remote Hierarchy**. But it is not tested.

1.3.2 NG Remote Inspector

Equivalent of **Inspector** window in **Unity Editor**. An **NG Remote Inspector** is associated to a **NG Remote Hierarchy**. From there, you can change any value of a *GameObject* or a *Component* and see the result on your device. It supports materials and shaders editing as well.

Inspector: Display almost the same content as **Unity**'s Inspector in the same way.

Batch Mode: Most packets affecting *Component* or *GameObject* are batched. Batches can be saved and reapplied at any time. (They are not persistent! If you compile or restart **Unity** they will disappear.)

Historic: Displays packets sent to the server.

It supports almost all features of *Inspector* window:

- **Active**: A checkbox to toggle *GameObject*'s active state.
- **Name**: An input to change *GameObject*'s name.
- **IsStatic**: A checkbox to toggle *GameObject*'s static state.
- **Tag**: A popup to change *GameObject*'s tag. Look at **NG Remote Hierarchy**'s options to change the behaviour.
- **Layer**: A popup to change *GameObject*'s layer.
- **List all Component**: A list of every Component and their fields.
    - Almost all types are supported: *Char*, *Byte*, *SByte*, *Int16*, *Int32*, *UInt16*, *UInt32*, *Single*, *Double*, *String*, *UnityEngine.Object*, array, *List<>*, struct and class. *Int64* and *UInt64* are supported but only in **Unity 5**. *Decimal* is not supported, since **Unity Editor** does not support it at all.

- **Drag & Drop Object**: Allows to drag & drop *GameObject* from **NG Remote Hierarchy** into *Object* field, and also between fields. Note that because the type might be unknown, you might be able to drag *Object* into any fields without harming the build.
- **Copy Component**: Copies all values of a *Component*, like **Unity**'s *Inspector.* It requires the *C# Type* to be present in your project!
- **Delete Component**: Delete a *Component*, like **Unity**'s *Inspector.*
- **Materials**: List all materials and their shaders' properties.

Material & Shader

From **NG Remote Inspector** you can change a Material's shader. The shaders available are those embarked into the build by **Unity**. You can add your own by pressing the button "Scan" at the right of the field "Shaders" in the script *NG Remote Scene*. It will embed all the shaders available in your project into the build.

1.3.3 NG Remote Project

Embed any assets you want to test. In normal case, **Unity** would embed the bare necessities only.

In field "NG Remote Project Assets" of *NG Server Scene*, you can force which assets you want to add to the build. After you have chosen all the assets, click on "References Resources".

When connected with the remote scene, connect with the button "Connect" on the top-right of **NG Remote Project**.

Options

- **Auto Load**: It loads the embed assets as soon as it can. Disable it to avoid unwanted packet from **NG Remote Project**.

1.3.4 NG Remote Camera

Camera: Move through your entire scene as if you were moving in a scene window. Pick which Camera you want to see through. When watching from the ghost camera, you can anchor it to any collider on the scene, to make it follow it. (A car, a plane, a bullet, a player, etc...)

Click on the picture to ask for a raycast and the server will reply with a list of colliders. It is pretty useful when you want to select a particular Game Object on the scene without having to look for it in **NG Remote Hierarchy**, because the list is pingable.

Modules: List modules used by **NG Remote Camera**. A module represents a data transferred between the build and your **Unity**. It can be disable if you don't want it to run. There is 4 built-ins data modules. The main for the image, one for the touch, the keyboard, the mouse. You can implement your own for any controller or metrics or gameplay data.

The data received is saved and used by **NG Replay**. From **NG Replay** you can see your data (You display it the way you want), it is particularly useful for a bug report, as you have the video and the inputs for each frame.

Export Replay: Export your current feed to **NG Replay**.

Connect: Open a connection with the ghost camera.

See: How to implement a CameraDataModule for **NG Remote Camera**.

Options

- **Record Duration**: Last seconds recorded to export to **NG Replay**.
- **Target FPS**: Number of images per second requested to the build. Gauge yourself if your mobile can handle more or less.
- **Camera Settings**: The settings used by the ghost camera in your scene.

1.3.5 NG Replay

Save and watch your replays as much as you want with **NG Replay**. Start a feed with **NG Remote Camera**, export the feed to **NG Replay** and analyze what happened on your scene. Like a video, you can play it or watch it frame by frame, change the speed, toggle modules, see what keys were pressed or where was the mouse on the screen.

Save a replay and share it with anyone who possess **NG Tools**.

1.4 NG Draggable Object

Replaces the default drawer for any **Unity**'s *Object*. It grants you the ability to drag & drop in the **Inspector** an *Object* from a field to another field.

NG Draggable Object is smart, when you drag a *GameObject* into a field, it will display you all its *Component* matching the field's type instead of assigning the first one like **Unity** does.

Furthermore, when you drag a *Component* into a field with an unmatching type, it applies the behaviour described just previously.

When there is more than one *Component* matching the field's type, a label "#N" is display at the left of the field, with N the position of the *Component* in the *GameObject*. Right-click on the field to switch to another *Component* from the same *GameObject*.

See: Look at **NG Draggable Object**'s settings in **NG Settings**.

1.5 NG Fav

Saves your selections in **NG Fav**. Any *GameObject* or asset can be saved, even runtime *GameObject*!

Runtime *GameObject* defines an entity that you create after pressing Play. A hero, some bullets, an item, a path, a mountain, a card, etc...

When you stop playing, the runtime *GameObject* is destroyed, so is its reference in **NG Fav**. But there is a simple way to remember the reference, you must implement the interface *ICustomFavorite* in one script of the *GameObject*.

Each selection can be pinged or selected by clicking on it or by pressing the shortcut (For the first 10 selections, Shift + F1 to F10).

Add new selection by right-clicking on any *GameObject* or any assets and click on "Add Selection". Else, drag & drop your selection into **NG Fav** to create a new one, or drop it on an existing selection to append to it.

NG Fav handles multi-list of favorites. Use it to have a specific list of favorites between projects and scenes or any other context.

See: How to implement an ICustomFavorite for **NG Fav**.

See: Look at **NG Fav**'s settings in **NG Settings**.

1.6 NG Inspector Gadget

A gathering of tiny tools, to slightly improve your experience in **Unity**'s **Inspector**.

1.6.1 Reorder Components

Adds the entry "Reorder Components" in context menu of *Component* in **Inspector**.

Click on it to open a little wizard allowing you to easily reorder *Component* in the *GameObject*.

1.6.2 Clone Component

Adds the entry "Clone Component" in context menu of *Component* in **Inspector**.

Click on it to clone the current *Component* and append the new one rightafter the current one.

1.7 NG Prefs

Changes any values from *EditorPrefs* or *PlayerPrefs*.

Add, change and delete any keys.

Display a list of all data, filter by keys, refresh the whole list or clear them all!

Implement your own Prefs manager if you have particular requirements when it is about Prefs.

See: How to implement a PrefsManager for **NG Prefs**.

1.8 NG Hierarchy Enhancer

Whenever you hover over a *GameObject* with your mouse in *Hierarchy* window, an **NG** icon appears. Pass over it to deploy a menu.

The menu will call the method OnHierarchyGUI() from each script on the *GameObject*.

The menu dynamically generates GUI regarding scripts on it. The purpose of that is to display, change or action anything you want from Hierarchy without selecting the *GameObject* and then alter the value through the **Inspector**. Like a shortcut.

You can color the background of your *GameObject* and append an icon to it by changing the layers' color and texture in **NG Settings**. The background will effectively be drawn only if the alpha is higher than 0! Do not forget to set the alpha.

See: Look at **NG Hierarchy Enhancer**'s settings in **NG Settings**.

See: How to implement Hierarchy GUI for **NG Hierarchy Enhancer**.

1.9 NG Nav Selection

Any selection you do from **Hierarchy** and **Project** windows are memorized.

Buttons "Previous" and "Next" on your mouse allow you to switch the current selection. Unfortunately this feature is **only available on Windows**.

Note: This is different from *Undo* and *Redo*. In fact, if you *Undo* in **Unity**, you will cancel the last thing you have done, if it was a selection, you will switch to your penultimate selection, for everything else, it will be canceled. **NG Nav Selection** only focus on the selection.

See: Look at **NG Nav Selection**'s settings in **NG Settings**.

1.10 NG Hub

NG Hub gathers shortcuts. It contains several built-ins tools to achieve your tasks in a faster manner.

Tasks like opening Player, Physics, Quality windows, having the **Time**.*scale* always on screen, shortcuts to any assets or scenes.

By default, **NG Hub** is a simple window, but it can be docked at the top of your **Unity Editor**. This way, you don't have to bother yourself to put it in any window and you will finally use a wasted place, don't you think?

Right-click to open the contextual menu. From there you can edit/add new components or dock/undock **NG Hub**.

As always, you can implement your own component if you have specific needs.

See: How to implement HubComponent for **NG Hub**.

1.11 NG Assets Finder

Look for all references of an asset in your scene or your project. The result gives a very accurate position of the references, in which asset or *GameObject*, the field, the position in the array, if nested or not.

How To Use: Pick an asset in the *Object* field "Find Asset". Set the options and click on "Search References".

How To Use: Click on the button at the left of the field "Find Asset" to toggle replacement. 2 buttons will appear at the bottom. "Replace References" will only replace the reference if the reference is the one in "Find Asset". "Set all References" will set all the references.

The options are quite simple.

- Click on "Scene" to enable/disable the search in the current scene.
    - List of *GameObject*. Drag & drop *GameObject* in this area to narrow down the search in the scene.
- Click on "Project" to enable/disable the search in the project.
    - You have 3 mains options:
        - "Asset": Search through any asset except prefabs and scene.
        - "Prefab": Search into prefabs.
        - "Scene": Search into scenes. This option is disabled when *Object* in "Find Asset" is in the scene.
    - List of folders. Drag & drop files or folders in this area to narrow down the search in the project.

Tip: Right-click on any asset in **Project** and click on "Search Asset".

Tip: Right-click on any *GameObject* in **Hierarchy** and click on "Search Game Object".

Tip: Right-click on any *Component* in **Inspector** and click on "Search Component".

1.12 NG Shader Finder

Look for any *Material* using a given *Shader*. This is useful to know who is using a *Shader* or if you want to replace a *Shader* by a new one.

How To Use: Pick a *Shader* in the *Object* field "Find Shader" and click on "Search all Material".

How To Use: Click on the button at the left of the field "Find Shader" to toggle replacement. 2 buttons will appear at the bottom. "Replace" will only replace the reference if the reference is the one in "Find Shader". "Set all" will set all the references.

Tip: Right-click on any *Material* or *Shader* in **Project** and click on "Search Material using this Shader".

1.13 NG Components Inspector

Compare 2 *Component* side by side.

How To Use:

- Right-click on any *Component* in **Inspector** and click on "Add to NG Components Inspector". *Component* will be saved in a historic.
- Left-click on any entity of the historic to set the left *Component*.
- Right-click on any entity of the historic to set the right *Component*.

- The button "No custom Editor" will force the use of a default rendering instead of an *Editor* if there is one.

1.14 NG Missing Script Recovery

Recover missing scripts from a *GameObject* or in your entire project. Note that this tool can only works on prefab.

When fixing missing scripts, **NG Missing Script Recovery** digs in the prefab's file to find fields to match with a potential *Type*. In the worst case, there will be no field at all, thus the missing script can not be recovered.

Potential **Type** are displayed with their matching score, score relying on matching fields.

NG Missing Script Recovery contains an automatic recovery. Inside tab **Project** in the window, you can click on **Scan** to look for every missing scripts in your project. You can trigger the automatic recovery to let **NG Missing Script Recovery** recovers each case it can.

Automatic recovery contains the following options:

- **Recovery Mode**: **Automatic** or **Manual**
    - **Automatic** will let **NG Missing Script Recovery** automatically recovers each case when one perfect match is found.
    - **Manual** will let you recover each case, one by one.
- **Promp On Pause**: When automatic recovery is triggered, **Unity** will pop an alert when a case can not be fixed.
- **Use Cache**: Each fix is cached and will be use by the automatic recovery to manage similar missing scripts.
- **Supa Fast**: When recovering, **Unity** won't display any feedback and will focus more on completing the recovery as fast as possible.
- **Recovery Log File**: Path of file where the recovery will be logged. You can find all fixes done by automatic recovery **ONLY**, manual fixing is not logged!

1.15 NG Sync Folders

Synchronize slave folders with a master folder.

This is useful when working on multiple copies of a project.

Master Folder: The master defines the primary folder on which each slave must be synchronized.

Relative Path: A relative path can be append on each master/slave folder. Use it to work relatively from folders. e.g set master with C:\abc\def\UnityProject, relative path with Assets\Scenes or else. This avoid the process of potentially a lot of unwanted files/folders.

Slaver Folder: Slave defines the folder that must match the master.

Inclusive Filters: If there is at least one filter, each file matching an inclusive filter will be included, others are rejected. If there is none, filtering is not applied.

Exclusive Filters: Each file matching an exclusive filter will be discarded.

Use Cache: Scanned files are cached, use it for efficiency.

1.16 NG Component Replacer

Tool not finished yet. Maybe one day.

1.17 NG Renamer

Rename multiple files all at once.

How To Use: You can select assets in **Hierarchy** or **Project**.

How To Use: Drag & drop any assets or even folders from Explorer/Finder.

NG Renamer contains many filters to rename assets.

See: How to implement TextFilter for **NG Renamer**.

See: Refer to the class *TextFilter* for more informations.

1.18 NG Settings

All settings from all tools in **NG Tools** are gathered in **NG Settings** which saves almost everything in *EditorPrefs* or in an asset in the project. It is the equivalent of **Unity**'s *Preferences*, but more advanced and modular.

Using the asset file, it is automatically exportable. Also, because it lives in an asset, you can easily create many settings files and switch between them seamlessly.

- **NG Console**
    - **Fine-grained settings** Almost all settings in **NG Console** can be changed.
    - **Fine-grained editors' extensions** You can choose which executable must open which extension.
    - **Themes** Overwrites visual settings in your settings.
    - **Presets** Overwrites behaviour settings in your settings.

See: How to implement your own settings in **NG Settings**.

1.19 Additional Tools

Here will be described independent tools not relying on any of the previous tools above.

1.19.1 ScriptableObject Creator

If you give attention to details, you might have noticed the new entry "Scriptable Object" in the menu Assets/Create.

Often people ask for a tool to easily create *ScriptableObject* asset, and there is none embedded in **Unity**.

So! Here it comes! A friendly and easy to use wizard to generate your assets!

1.19.2 GroupAttribute

Regroup fields sharing the same group name into a single place in **Inspector**.

How To Use: Apply the attribute *GroupAttribute* on any field. It accepts 2 arguments, the first is the group name and the second defines if the field is hidden. The latter is used only in a specific case.

An example of *GroupAttribute*:

```
using NGTools;

public class TestGroupAttribute : MonoBehaviour
{
    [Group("Group 1")]
    public string   aString;
    [Group("Group 1")]
    public int      anInteger;
}
```

If you want to group a field having a CustomPropertyDrawer, you must use the attribute *InGroupAttribute*. It takes the group name as sole argument.

Note: A field using *InGroupAttribute* can not be the first field of its group in the *MonoBehaviour*! Only a *GroupAttribute* can do that, for this specific case there is only one workaround.

An example of *InGroupAttribute*:

```
using NGTools;
using UnityEngine.Events;

public class TestInGroupAttribute : MonoBehaviour
{
    [Group("InGroup 1", true)]
    public bool unused;
    [InGroup("InGroup 1")]
    public UnityEvent       anInteger;
}
```

You have to add a dummy field before the concerned *InGroupAttribute* and assign on it a *GroupAttribute* with "true" as second argument. This will hide the field "unused".

Note: The group name accepts rich text. Look at **Unity** documentation about Rich Text at https://docs.unity3d.com/Manual/StyledText.html.

Note: Attribute *Header*, *Range* and *Space* will not work when using attribute *Group*, you must use their equivalent *GHeader*, *GRange* and *GSpace*. You can create your own.

1.19.3 NG Check GUID

Fetch a GUID from any asset or an asset from its GUID.

How To Use: Drop an asset in the *Object* field or pick an asset directly from the field.

How To Use: Write a valid GUID to fetch its asset.

Note: Copy a GUID in the clipboard and go into **NG Check GUID**, it will automatically paste it in the list.

2 Technical Usage

2.1 NG Console

2.1.1 Module

Class *Module* is the backbone of **NG Console**, allowing to create a visible module (Selectable through the header menu) or an invisible module able to do a background task.

- **MainModule** is the main module, can not be more obvious. It displays logs as the native console with useful features aside.
- **ArchiveModule** is visible and interferes with other modules. It allows to save logs and assigns them a note.
- **ColorMarkersModule** is an invisible module, but it manually adds a button in the header menu. It colors background of log regarding markers' settings.

How to implement a *Module*:

- Create a class inheriting from *Module*.
- Add attribute *Serializable*.
- [Optional] Add attribute *VisibleModule* if you want your module to be exposed and visible.

An example of *Module*:

```
using NGToolsEditor.NGConsole;
using System;
using UnityEngine;

[Serializable]
[VisibleModule(70)]
public class TestModule : Module
{
    public  TestModule()
    {
        this.name = "test";
    }

    public override void    OnGUI(Rect r)
    {
        GUI.Button(r, "Button");
    }
}
```

Note: **NG Console** contains 4 (+1) built-ins *Module*: *MainModule*, *ArchiveModule*, *ColorMarkersModule* and *RemoteModule* (and *DebugModule*).

See: Refer to the class *Module* for more informations.

2.1.2 StreamLog

StreamLog simply displays logs. By default, it can filter by **Unity**'s log types (Normal, Warning, Error and Exception) and can be deeply modified by your own filters which can change the stream of displaying logs.

The class can be overridden to display your own stream.

MainStream is used by *MainModule*. The only feature it provides is the *Collapse* option synchronized with the native console. This stream can not be deleted.

CompilerStream is also used by *MainModule*. It appears when compiler outputs warnings or errors and disappears when there is none. This stream can not be deleted.

2.1.3 Row

NG Console invokes its 2 events *CheckNewLogConsume* then *PropagateNewLog* for each new log.

The first event does not handle the incoming log but check if it is consumed or if it must be consumed. A *StreamLog* can chose to discard logs that are consumed, depending on its purpose.

For example, *MainStream* and *CompilerStream* do not care of about consumption, they accept all logs. Nevertheless, *StreamLog* does. When a log is consumed, all next *StreamLog* in the list will reject it. That way, you can have a *StreamLog* dedicated to memory, AI, debugs, warning or initializations, etc...

The second event effectively handles the log regarding what happened just before in the previous event.

How to implement a Row:

- Create a class inheriting from *Row.*
- Add attribute *Serializable*.
- [Optional] Add attribute *RowLogHandler* if you want your implementation be called when a new log arrives. (It is possible you do not want it to be called. Like *RemoteRow*, a special *Row* implemented and used only by *RemoteModule* to handle asynchronous *Row* from network.)
    - If *RowLogHandler* is present, you must implement the following method:

```
    private static bool CanDealWithIt(UnityLogEntry log)
    {
        return true; // Here is the condition of your Row, checking if it should handle the log.
    }
```

An example of *Row*:

```
using NGToolsEditor.NGConsole;
using System;
using UnityEditor;
using UnityEngine;

[Serializable]
[RowLogHandler(1000)]
public class TestRow : Row
{
    private static bool CanDealWithIt(UnityLogEntry log)
    {
        return log.condition.Contains("test");
    }

    public override float   GetWidth()
    {
        return 0F;
    }

    public override float GetHeight()
    {
        return EditorGUIUtility.singleLineHeight;
    }

    public override void    DrawRow(RowsDrawer rowsDrawer, Rect r, int i, bool?
collapse)
    {
        float   originWidth = this.editor.position.width -
rowsDrawer.verticalScrollbarWidth + rowsDrawer.scrollPosition.x;

        r.x = rowsDrawer.scrollPosition.x;
        r.width = originWidth;
        r.height = this.editor.settings.log.height;

        // Draw background when focus, even or odd.
        this.DrawBackground(rowsDrawer, r, i);

        // Handle default log focus.
        this.HandleDefaultSelection(rowsDrawer, r, i);

        GUI.Label(r, "[Test]", this.editor.settings.log.style);
    }
}
```

To test it, just write a log containing the word "test" or from a file with the word "test" in its path. A simple text "[Test]" should appear.

Note: **NG Console** contains 6 built-ins *Row*: *DefaultRow*, *CompileRow*, *MultiContextRow*, *DataRow*, *RemoteRow* and *BlankRow*.

See: Refer to the class *Row* and *UnityLogEntry* for more informations.

2.1.4 ILogFilter

ILogFilter checks whether or not a log matches a condition. It is most often used through *GroupFilters*, a class providing an easy way to handle multiple filters. It is used by *StreamLog* to filter incoming logs and *ColorMarkersModule* to colorize log's background.

How to implement a filter:

- Create a class implementing the interface *ILogFilter.*
- Add attribute *Serializable*.
- Define your condition in method CanDisplay().
- If you want your filter to be dynamic, you can implement GUI in OnGUI().
- Method ContextMenu() is used by some *Row* implementing a context menu. It allows your filter to interact with logs.
- Create a file named after the filter's class name in the folder Locale/english. Add it the pair "XXXX=YYYY" with XXXX the exact string of the filter's class name and YYYY a human readable version in english.

An example of a filter:

```
using NGToolsEditor.NGConsole;
using System;
using UnityEditor;
using UnityEngine;

[Serializable]
public class TestFilter : ILogFilter
{
    public bool Enabled { get; set; }

    public event Action ToggleEnable;

    private string  keyword;

    public FilterResult CanDisplay(Row row)
    {
        if (row.log.condition.Contains(this.keyword) == true)
            return FilterResult.Accepted;
        return FilterResult.None;
    }

    public void OnGUI()
    {
        this.keyword = EditorGUILayout.TextField("Filter test keyword: ",
this.keyword);
    }

    public void ContextMenu(UnityEditor.GenericMenu menu, Row row, int position)
    {
        menu.AddItem(new GUIContent("Menu Test"), false, () => Debug.Log("Hello
there!"));
    }
}
```

Note: **NG Console** contains 5 built-ins *ILogFilter*: *ContentFilter*, *HierarchyFilter*, *MaskTypeFilter*, *NameHierarchyFilter* and *TagFilter.*

See: Refer to the interface *ILogFilter* and class *GroupFilters* for more informations.

2.1.5 IStackFrameFilter

IStackFrameFilter allows to remove a frame from the stack trace based on the raw frame given by **Unity**'s log.

This interface gives an easy way to create your own *Debug.Log*().

An example of a *IStackFrameFilter*:

```
using NGToolsEditor.NGConsole;

public class UselessFrameFilter : IStackFrameFilter
{
    public bool Filter(string frame)
    {
        return frame.StartsWith("YourDebugClass:") == true ||
                frame.StartsWith("AnotherDebugClass:WithSpecificMethod") ==
true;
    }
}
```

To test, change the strings by your own debug class. Anytime you will open a stack trace, your class will not be in there.

Note: **NG Console** contains 1 built-in *IStackFrameFilter*: *UselessFrameFilter.*

See: Refer to the interface *IStackFrameFilter* for more informations.

2.1.6 ILogExporter

ILogExporter allows to export logs in your own format.

How to implement a log exporter:

- Create a class implementing the interface *ILogExporter.*
- I strongly recommend you to look at existing implementation of *ILogExporter.* It might be rough, but the implementations cover good cases.
- Please accept my apologies about this short todo.

Note: **NG Console** contains 2 built-ins *ILogExporter*: *RawExporter* and *XMLExporter.*

See: Refer to the interface *ILogExporter* for more informations.

2.1.7 Editor Openers

An opener is just an alternative way to open your asset. It gives the possibility to associate a set of extensions with a software.

It falls back on **Unity** if the asset is not handle by an opener from **NG Tools**.

To modify those, you must change **NG Settings** -> General -> General -> Editor Extensions.

How to implement an opener:

- Create a class implementing *IEditorOpener.*

- The method CanHandleEditor() checks if the given editor's path is handled. Notice that you only have the path of the software, this is the only data provided to the opener.
- In addition of the method CanHandleEditor(), the property "defaultArguments" is used when CanHandleEditor() returns true. CanHandleEditor() is called when you choose an executable in Editor Extensions in **NG Settings** as explained just above.
- Look at the implementation of *NotepadPlusPlusOpener* to see a very simple opener.

Note: **NG Console** contains 6 built-ins *IEditorOpener*: *MonoDevelopOpener, VisualStudioOpener, NotepadPlusPlusOpener, SublimeOpener, VimOpener* and *EmacsOpener.*

See: Refer to the interface *IEditorOpener* for more informations.

2.1.8 Preset

Preset grants you the possibility to change in an instant all settings about **NG Console**'s behaviour.

All implementations of *Preset* are displayed in section "Presets" in **NG Settings**.

Note: **NG Console** contains 3 built-ins *Preset*: *FastestPreset*, *MinimalPreset* and *VerbosePreset*.

See: Refer to the class *Preset* for more informations.

2.1.9 Theme

Theme allows you to create your own predefined custom style.

Allowing to alter the header menu, the visual of a log and its stack trace, the display of the source code and many other tiny details.

All implementations of *Theme* are displayed in section "Themes" in **NG Settings**.

Note: **NG Console** contains many built-ins *Theme*: *DarkTheme*, *LightTheme*, and alll FontSizeXXTheme.

See: Refer to the class *Theme* for more informations.

2.2 NG Game Console

2.2.1 NG Game Console

2.2.1.1 DataConsole

Displays a data on the console, anything you want from the scene, *GameObject*, gameplay, rules, statistics, etc...

How to implement a *DataConsole*:

- Create a class inheriting from *DataConsole*.
- Override methods CanDrawShortGUI() and ShortGUI() if you want a short version of your data.

- Override methods CanDrawFullGUI() and FullGUI() if you want a complet version of your data.
- Drop your script on a *GameObject*.
- Add the instance of your *DataConsole* in field "Data Console" of **NG Game Console** in **Inspector** window.

An example of a *DataConsole*:

```
using NGTools.NGGameConsole;
using UnityEngine;

public class TestData : DataConsole
{
    public override void    ShortGUI()
    {
        GUILayout.Label("Short Test");
    }

    public override void    FullGUI()
    {
        GUILayout.Label("Full Test");
    }
}
```

2.2.2 NG CLI

2.2.2.1 Root command

A root command is simply a command with sub-commands. **NG CLI** requires root commands to execute anything.

A command triggers an action, change a variable, call a function or anything else.

How to implement root command:

- Create script inheriting from *MonoBehaviour.*
- Create properties or methods with the attribute *CommandAttribute*.
- Drop your script on a *GameObject*.
- Add the instance of your script in field "Root Commands" of *NGCLI* in **Inspector** window.

Note: The attribute *CommandAttribute* can be applied on *public* properties, *static* or not, with a getter and a setter.

Note: The attribute *CommandAttribute* can be applied on *public* methods, *static* or not, with a "string" return type. *CommandAttribute* only supports primary types, *decimal* and *string* as arguments.

An example of a root command:

```
using NGTools.NGGameConsole;
using UnityEngine;

public class TestRootCommand : MonoBehaviour
{
```

```
    [Command("instanceInteger", "")]
    public int a { get; set; }
    [Command("instanceString", "")]
    public string b { get; set; }
    [Command("instanceFloat", "")]
    public float c { get; set; }
    [Command("instanceBoolean", "")]
    public bool d { get; set; }
    [Command("staticDecimal", "")]
    public static decimal e { get; set; }
    [Command("staticInt", "")]
    public static int f { get; set; }
    [Command("staticString", "")]
    public static string g { get; set; }
    [Command("staticFunction", "")]
    public static string Fn1()
    {
        return "A lambda result.";
    }
    [Command("staticFunctionWithArgument", "")]
    public static string Fn2(int b)
    {
        return "Fn2(" + b + ")";
    }
}
```

You can use commands to interact with your game, to change your configuration, stop the timer, save the state of anything, change scene, delete things, start a song, dance salsa, eat tacos and more and more!

2.3 NG Remote Scene

2.3.1 TypeHandler

The class *TypeHandler* gives the possibility to implement your own way to serialize and deserialize a given type, choosing which fields or properties you want to transmit via network.

Basically, a custom class/struct gets all its public fields automatically serialized/deserialized; but if you want it to be more efficient or just want to discard fields from the serialization pass, you might need to implement a *TypeHandler.*

Any implementation of *TypeHandler* needs to be assigned the attribute *PriorityAttribute* to define an order between *TypeHandler.* Additionally, each *TypeHandler* must have its implementaton of *TypeHandlerDrawer.*

Here is the implementation of the built-in *BooleanHandler*:

```
using System;

namespace NGTools.NGRemoteScene
{
    [Priority(0)]
    public class BooleanHandler : TypeHandler
    {
        public override bool    CanHandle(Type type)
        {
```

```
            return type == typeof(Boolean);
        }

        public override void    Serialize(ByteBuffer buffer, Type fieldType,
object instance)
        {
            buffer.Append((Boolean)instance);
        }

        public override object      Deserialize(ByteBuffer buffer, Type
fieldType)
        {
            return buffer.ReadBoolean();
        }
    }
}
```

Note: **NG Remote Scene** contains built-ins for all primary types from .Net (Boolean, Byte, SByte, Char, Int16, Int32, Int64, UInt16, UInt32, UInt64, Single, Double), plus complex types (Array, enum, class, struct, string).

Note: **NG Remote Scene** also contains built-ins for some types from UnityEngine (Object, Rect, Quaternion, Vector2, Vector3, Vector4, Color).

See: Refer to the class *TypeHandlerDrawer* and *PriorityAttribute* for more informations.

2.3.2 TypeHandlerDrawer

When you implement a *TypeHandler*, you need to implement a *TypeHandlerDrawer.*

The drawer is the equivalent of *PropertyDrawer* from **Unity**. It draws the type in **NG Inspector**, but it has to take care of few things.

TypeHandlerDrawer is a bit more complex, because **NG Remote Scene** has to manage network, meaning the value you are watching might be obsolete, and any change you do is asynchronous.

Therefore, the drawer has to display feedbacks and informations about the value for the sake of the user experience. But lucky you, it is already done.

Also, the class *TypeHandlerDrawer* needs the attribute *TypeHandlerDrawerForAttribute* to be linked to a *TypeHandler.*

Here is the implementation of the built-in *BooleanDrawer* with comments (Don't be afraid! It is not that big, just the comments make it feel bigger):

```
using NGTools;
using System;
using UnityEditor;
using UnityEngine;

namespace NGToolsEditor.NGRemoteScene
{
    // Which TypeHandler it is related.
    [TypeHandlerDrawerFor(typeof(BooleanHandler))]
    public class BooleanDrawer : TypeHandlerDrawer
    {
```

```
        // An animator displaying a smooth animation when a new value is fetched
from the server.
        private BgColorContentAnimator  anim;
        // Because the update is asynchronous, it displays the last value
assigned.
        // Moreover, it handles dragging the field, preventing the value to
shake when the server updates while you are dragging (For integer and float).
        private ValueMemorizer<Boolean> drag;

        public  BooleanDrawer(TypeHandler typeHandler) : base(typeHandler)
        {
            this.drag = new ValueMemorizer<Boolean>();
        }

        public override void    Draw(Rect r, DataDrawer data)
        {
            if (this.anim == null)
                this.anim = new BgColorContentAnimator(data.inspector.Repaint,
1F, 0F);

            // A notification is generated when the server sends the new value.
            if (data.inspector.hierarchy.GetUpdateNotification(data.GetPath())
== true)
            {
                this.drag.NewValue((Boolean)data.value);
                this.anim.Start();
            }

            using (this.anim.Restorer(0F, .8F + this.anim.Value, 0F, 1F))
            {
                EditorGUI.BeginChangeCheck();
                // Draws the field with the current working value (Remember the
drag story!).
                Boolean newValue = EditorGUI.Toggle(r,
ObjectNames.NicifyVariableName(data.name), this.drag.Get((Boolean)data.value));
                if (EditorGUI.EndChangeCheck() == true)
                {
                    this.drag.Set(newValue);
                    this.AsyncUpdateCommand(data.unityData, data.GetPath(),
newValue, typeof(Boolean));
                }

                // Display last value if a new request is pending.
                this.drag.Draw(r);
            }
        }
    }
}
```

See: Refer to the class *TypeHandler* and *TypeHandlerDrawerFor* for more informations.

2.3.3 ComponentExposer

Sometimes, it happens that you specifically need to discard a field or a property from being serialized or even reached.

An easy example would be the class *MeshRenderer*. Whenever a *MeshRenderer* is serialized, it would extract all fields and properties including the property "material", which would leads to an unwanted instance of the material (Because when you make a call to this property, **Unity**

automatically creates a copy of the *Object*). It would be the same for the class *MeshFilter* and its property "mesh".

To avoid this unfortunate issue, *ComponentExposer* provides 2 methods you might override to explicitly expose fields and properties.

Also, your class must have the attribute *ComponentExposingForAttribute* to define which type it is related to.

Here is the implementation of the built-in *MeshRendererExposer*:

```
using System;
using System.Reflection;
using UnityEngine;

namespace NGTools.NGRemoteScene
{
    [ComponentExposingFor(typeof(MeshRenderer))]
    public class MeshRendererExposer : ComponentExposer
    {
        public override PropertyInfo[] GetPropertyInfos(Component component)
        {
            PropertyInfo[]  fields = new PropertyInfo[7];

            Type    type = component.GetType();

            fields[0] = type.GetProperty("enabled");
            fields[1] = type.GetProperty("shadowCastingMode");
            fields[2] = type.GetProperty("receiveShadows");
            // The field "material" is explicitly not included!
            fields[3] = type.GetProperty("sharedMaterials");
            fields[4] = type.GetProperty("useLightProbes");
            fields[5] = type.GetProperty("reflectionProbeUsage");
            fields[6] = type.GetProperty("probeAnchor");

            return fields;
        }
    }
}
```

Note: **NG Remote Scene** contains 7 built-ins *ComponentExposer*: *AudioSourceExposer*, *ColliderExposer*, *MeshFilterExposer*, *RendererExposer*, *RigidbodyExposer*, *TrailRendererExposer* and *TransformExposer*.

See: Refer to the class *ComponentExposingForAttribute* for more informations.

2.3.4 ArgumentDrawer

NG Inspector allows to invoke method of any *Component* directly through in **NG Remote Inspector**.

But you might have complex arguments in your methods, hence *ArgumentDrawer*.

ArgumentDrawer draws editor GUI for a specific type, like *PropertyDrawer*.

Your implementation of *ArgumentDrawer* must have the attribute *ArgumentDrawerForAttribute* to be linked to a type.

Here is the implementation of the built-in *BooleanArgumentDrawer*:

```
using System;
using UnityEditor;

namespace NGToolsEditor.NGRemoteScene
{
    [ArgumentDrawerFor(typeof(Boolean))]
    public class BooleanArgumentDrawer : ArgumentDrawer
    {
        public BooleanArgumentDrawer(string name) : base(name, typeof(Boolean))
        {
        }

        public override void    OnGUI()
        {
            this.value = EditorGUILayout.Toggle(this.name, (Boolean)this.value);
        }
    }
}
```

Note: **NG Remote Scene** contains built-ins for all primary types from .Net (Boolean, Byte, SByte, Char, Int16, Int32, Int64, UInt16, UInt32, UInt64, Single, Double), plus only one complex type (String).

Note: **NG Remote Scene** also contains built-ins for some types from UnityEngine (Rect, Quaternion, Vector2, Vector3, Vector4, Color).

See: Refer to the class *ArgumentDrawerForAttribute* for more informations.

2.3.5 CameraDataModule

CameraDataModule allows to communicate any data between the server and the client in real time.

The implementation of a *CameraDataModule* is complex due to the creation of 3 different classes.

One deriving from *CameraServerDataModule*, this class lives in the server-side and handles the data to send to the client. You basically have 4 methods, Awake(), OnDestroy, OnGUI() and Update().

Override OnGUI() or Update() to pack your data and send it to the client.

Override Awake() or OnDestroy() to initialize stuff, like managing incoming packets from the client.

Here is an implementation of a *CameraServerDataModule*:

```
using UnityEngine;

namespace NGTools.NGRemoteScene
{
    public class TestModule : CameraServerDataModule
    {
```

```
        public const byte   ModuleID = 255;
        // The higher, the bigger. But it can not be equal or higher than 1000.
        public const int    Priority = 1;
        public const string Name = "Test";

        private float       nextTime;

        public  TestModule() : base(TestModule.ModuleID, TestModule.Priority,
TestModule.Name)
        {
        }

        // Send a packet containing the current frame count every second.
        public void Update(ICameraScreenshotData data)
        {
            float   t = Time.time;

            if (t <= this.nextTime)
                return;

            this.nextTime = t + 1F;

            // Put the frame count into the buffer which is an integer of 32
bits.
            Utility.sharedBBuffer.Append(Time.frameCount);

            // You can choose to send to the sender.
            data.Sender.AddPacket(new CameraDataPacket(this.moduleID, Time.time,
Utility.sharedBBuffer.Flush()));

            // Or to broadcast to all clients.
            //data.TCPListener.BroadcastPacket(new
CameraDataPacket(this.moduleID, Time.time, Utility.sharedBBuffer.Flush()));
        }
    }
}
```

Here is an extended implementation if you wish to share settings with the client.

```
using UnityEngine;

namespace NGTools.Network
{
    public partial class PacketId
    {
        // This ID must be unique! Make sure it does not collide with any other
packet.
        public const int    Camera_ClientModuleSetAnySetting = 100000;
    }

    [PacketLinkTo(PacketId.Camera_ClientModuleSetAnySetting)]
    public class ClientModuleSetAnySettingPacket : Packet
    {
        public int  setting;

        protected   ClientModuleSetAnySettingPacket(ByteBuffer buffer) :
base(buffer)
        {
        }

        public  ClientModuleSetAnySettingPacket(int setting)
```

```
        {
            this.setting = setting;
        }
    }

    public class TestModule : CameraServerDataModule
    {
        [...]

        public int  anySetting;

        [...]

        public override void    Awake(NGServerScene scene)
        {
            // Tell the packet executer which method handles this ID when
receiving it.

scene.executer.HandlePacket(PacketId.Camera_ClientModuleSetAnySetting,
this.SetAnySetting);
        }

        public override void    OnDestroy(NGServerScene scene)
        {

scene.executer.UnhandlePacket(PacketId.Camera_ClientModuleSetAnySetting);
        }

        [...]

        private void    SetAnySetting(Client client, Packet _packet)
        {
            ClientModuleSetAnySettingPacket packet = _packet as
ClientModuleSetAnySettingPacket;

            // Override the current setting with the new one.
            this.anySetting = packet.setting;
        }
    }
}
```

The second class to implement is *CameraDataModuleEditor*, it lives in client-side in **NG Remote Camera** and handles data received from the server.

Here is an implementation of a *CameraDataModuleEditor*:

```
using UnityEngine;

namespace NGToolsEditor.NGRemoteScene
{
    public class TestModuleEditor : CameraDataModuleEditor
    {
        // This is the class you will use to store data for each frame.
        public class TestData : CameraData
        {
            // With any data you want inside.
            public int  frameCount;
        }

        // This is the settings you might want to share with the server.
```

```
        private int anySetting;

        public  TestModuleEditor() : base(TestModule.ModuleID,
TestModule.Priority, TestModule.Name)
        {
        }

        public override void    OnGUICamera(IReplaySettings settings, Rect r)
        {
            // Display a label over the camera feed in NG Remote Camera.
            EditorGUI.LabelField(r, "Setting " + this.anySetting);
        }

        public override void    OnGUIModule(NGHierarchyWindow hierarchy)
        {
            // Alter the setting and send it to the server.
            EditorGUI.BeginChangeCheck();
            this.anySetting = EditorGUILayout.IntField("Any Setting",
this.anySetting);
            if (EditorGUI.EndChangeCheck() == true &&
hierarchy.IsClientConnected() == true)
                hierarchy.Client.AddPacket(new
ClientModuleSetAnySettingPacket(this.anySetting));
        }

        public override void    OnServerInitialized(IReplaySettings settings,
Client server)
        {
            // Right after the server is connected, send a packet to set the
default setting.
            server.AddPacket(new
ClientModuleSetAnySettingPacket(this.anySetting));
        }

        public override void    HandlePacket(IReplaySettings settings, float
time, byte[] data)
        {
            // You must keep only the most recent data depending on the settings
you set.
            this.RemoveOldData(time - settings.RecordLastSeconds);

            // Put the data into a ByteBuffer to read it more easily.
            ByteBuffer  buffer = Utility.GetBBuffer();

            buffer.Append(data);

            this.data.Add(new TestData() {
                time = time,
                // Get the data from the buffer which was an integer of 32 bits.
                frameCount = buffer.ReadInt32()
            });

            Utility.RestoreBBuffer(buffer);
        }

        public override ReplayDataModule    ConvertToReplay(IReplaySettings
settings)
        {
            // The data must be converted into a ReplayDataModule to be used
properly by NG Replay. Look at the last class to implement.
            return new TestReplayModule(this);
```

```
        }
    }
}
```

The last class to implement is *ReplayDataModule*, it lives in client-side and is used by **NG Replay**.

Here is an implementation of a *ReplayDataModule*:

```
using NGTools;
using System;
using System.Text;
using UnityEditor;
using UnityEngine;

namespace NGToolsEditor.NGRemoteScene
{
    public class TestReplayModule : ReplayDataModule
    {
        public  TestReplayModule() : base(TestModule.ModuleID,
TestModule.Priority, TestModule.Name)
        {
        }

        public  TestReplayModule(TestModuleEditor module) :
base(TestModule.ModuleID, TestModule.Priority, TestModule.Name)
        {
            this.data.AddRange(module.data);
        }

        public override void    OnGUIReplay(Rect r)
        {
            if (this.index == -1)
                return;

            TestModuleEditor.TestData   data = this.data[this.index] as
TestModuleEditor.TestData;

            // Displays a label with the frame count on the top-left over the
video feed.
            r.x = 0F;
            r.width = 100F;
            r.height = 16F;
            EditorGUI.LabelField(r, "Setting " + data.frameCount);
        }

        public override void    Export(ByteBuffer writer)
        {
            writer.Append(this.data.Count);

            foreach (TestModuleEditor.TestData data in this.data)
            {
                writer.Append(data.time);
                writer.Append(data.frameCount);
            }
        }

        public override void    Import(Replay settings, ByteBuffer reader)
        {
            int count = reader.ReadInt32();

            this.data.Clear();
```

```
            this.data.Capacity = count;

            for (int i = 0; i < count; i++)
            {
                TestModuleEditor.TestData   input = new
TestModuleEditor.TestData();
                input.time = reader.ReadSingle();
                input.frameCount = reader.ReadInt32();
                this.data.Add(input);
            }
        }
    }
}
```

Note: **NG Remote Camera** contains 4 built-in *CameraDataModule*: *ScreenshotModule*, *KeyboardModule*, *MouseModule* and *TouchModule*.

See: Refer to the classes *CameraDataModule*, *CameraServerDataModule*, *CameraDataModuleEditor* and *ReplayDataModule* for more informations.

2.4 NG Fav

Favorites are saved using a simple thing, their path in the hierarchy.

When **NG Fav** tries to reconnect the reference, it searches through the hierarchy based on the whole path. If not found, it will look for the *GameObject* with the same name.

2.4.1 ICustomFavorite

NG Fav saves asset based on the path of it. It works for all cases except for runtime *GameObject*.

For that case **NG Fav** provides the interface *ICustomFavorite* to make the ephemeral reference a reliable reference.

How to implement *ICustomFavorite*:

- Implement the interface *ICustomFavorite* on a *MonoBehaviour.*
- Inside the method GetFavorite(), you need to set the arguments *identifier* and *resolver. resolver* must be assigned with a static function!
- Argument *identifier* is optional, but not *resolver.* Because the callback *resolver* can return a specific *GameObject*, therefore it does not require an *identifier.*
- Drop the class in the *GameObject* you want to put in favorite.

Callback *resolver* returns a *GameObject* based on the *identifier* you gave.

```
using UnityEngine;

namespace NGTools.NGFav
{
    public class Foo : MonoBehaviour, ICustomFavorite
    {
        void    ICustomFavorite.GetFavorite(out int identifier, out Func<int,
GameObject> resolver)
        {
```

```
            identifier = 0;
            resolver = Foo.ObjectResolver;
        }

        private static GameObject   ObjectResolver(int identifier)
        {
            // Return a GameObject based on the given identifier.
            // If you have a database, a factory, or from a save. Any identifier
that you might use to refer to a specific GameObject.
            return null;
        }
    }
}
```

Recalling runtime *GameObject* is truly useful when you have thousands of *GameObject* in *Hierarchy* window and you do not want to scroll over the whole list or use filter to look for your *GameObject*, and redoing that each time you restart your game.

2.5 NG Prefs

2.5.1 PrefsManager

Class *PrefsManager* allows to save, edit and delete data. Use it to implement your own system of preferences.

This class is nearly too simple and not enough advanced, it only supports int, string and float.

Here is the implementation of the built-in *EditorPrefManager*:

```
using NGTools;
using System;
using UnityEditor;
#if UNITY_EDITOR_OSX
using UnityEngine;
#endif

namespace NGToolsEditor.NGPrefs
{
    // This manager is pretty simple and straight. If you want to handle
encryption or else, you can do it here.
    public class EditorPrefManager : PrefsManager
    {
        public override void    DeleteKey(string key)
        {
            EditorPrefs.DeleteKey(key);
        }

        public override void    DeleteAll()
        {
            EditorPrefs.DeleteAll();
        }

        public override bool    HasKey(string key)
        {
            return EditorPrefs.HasKey(key);
        }

        public override float   GetFloat(string key, float defaultValue = 0F)
```

```
        {
            return EditorPrefs.GetFloat(key, defaultValue);
        }

        public override int     GetInt(string key, int defaultValue = 0)
        {
            return EditorPrefs.GetInt(key, defaultValue);
        }

        public override string  GetString(string key, string defaultValue =
null)
        {
            return EditorPrefs.GetString(key, defaultValue);
        }

        public override void    SetFloat(string key, float value)
        {
            EditorPrefs.SetFloat(key, value);
        }

        public override void    SetInt(string key, int value)
        {
            EditorPrefs.SetInt(key, value);
        }

        public override void    SetString(string key, string value)
        {
            EditorPrefs.SetString(key, value);
        }

        public override void    LoadPreferences()
        {
            try
            {
#if UNITY_EDITOR_WIN
                this.LoadFromRegistrar(@"SOFTWARE\Unity Technologies\Unity
Editor 5.x");
#elif UNITY_EDITOR_OSX

this.LoadFromRegistrar("/Users/Apple/Library/Preferences/com.unity3d.UnityEditor
" + Application.unityVersion.Substring(0, Application.unityVersion.IndexOf('.'))
+ ".x.plist");
#endif
            }
            catch (Exception ex)
            {
                InternalNGDebug.LogException(ex);
            }
        }
    }
}
```

Note: **NG Prefs** contains 2 built-ins *PrefsManager*: *PlayerPrefsManager* and *EditorPrefsManager.*

See: Refer to the class *PrefsManager* for more informations.

2.6 NG Hierarchy Enhancer

2.6.1 Hierarchy GUI

Draws GUI about your script directly in *Hierarchy* window.

An example:

```
private float OnHierarchyGUI(Rect r)
{
    r.xMin = r.xMax - 100F;
    if (GUI.Button(r, "What") == true)
    {
        Debug.Log("Else?");
    }
    return r.xMin;
}
```

Note: **NG Hierarchy Enhancer** draws thing from right to left, therefore the return value must be the leftmost position in X axis you consumed.

2.6.2 DynamicObjectMenu

Because you can not implement the method on **Unity**'s scripts. **NG Hierarchy Enhancer** provides *DynamicObjectMenu* to bypass this issue.

Each time you open the menu, all classes inheriting from *DynamicObjectMenu* are called and given the *GameObject* as argument. From there, it is up to you to make your shopping.

Note: **NG Hierarchy Enhancer** contains 1 built-in *DynamicObjectMenu*: *GameObjectMenu* (Allows you to toggle Active state, to manage AudioSource, Renderer and ParticleSystem).

See: Refer to the class *DynamicObjectMenu* for more informations.

2.7 NG Settings

Settings are saved in a ScriptableObject in an asset.

Modules can save their settings in this asset which is shareable.

- **NG Console** An example is done by **NG Console** which implements many sections in **NG Settings**.
    - **Themes** Implement your own theme in addition of the default Light and Dark themes based on **Unity Editor**.
    - **Presets** Like theme, you can implement your own predefined settings in addition of the default Fastest, Minimal and Verbose settings.

2.7.1 Custom settings

How to implement and integrate your own settings:

- First way:
    - Implement a class inheriting from *AbstractModuleSettings* with any settings you want.
    - In your module, add a field of type *AutoExposeSettings<T>* with T the class you just created.
    - In OnEnable() of your module, assign your field with an instance of *AutoExposeSettings<T>*.
    - In OnDisable(), add a call to Close() from your instance of *AutoExposeSettings<T>*.
    - In case you need to initialize GUI, you must override the method InitializeGUI() from *AbstractModuleSettings*. Of course this is optional, if you have your own way to initialize GUI.

This way is nice if you want to save your data separately from **NG Settings**. Your very own.

Here is the implementation of the first way: First, create the setting class:

```
using NGToolsEditor;
using UnityEngine;

public class DummySettings : AbstractModuleSettings
{
    public float    floatSetting = 70F;
    public string   stringSetting = "Test";
    public GUIStyle style;

    protected override void Reset()
    {
        base.Reset();

        this.floatSettingfloatSetting = 70F;
        this.stringSetting = "Test";
        this.style = null;
    }

    protected override void InitializeGUI()
    {
        if (this.style == null)
            this.style = new GUIStyle();
    }
}
```

Integrate your setting class through an instance of AutoExposeSettings:

```
using NGToolsEditor.NGConsole;
using System;
using UnityEditor;

[Serializable]
[VisibleModule(50)]
public class DummyModule : Module, IStreams
{
    [NonSerialized]
    private AutoExposeSettings<RemoteSettings>  settings;

    public override void    OnEnable(NGConsole editor, int id)
    {
```

```
        base.OnEnable(editor, id);

        this.settings = new AutoExposeSettings<DummySettings>("Dummy Module",
Utility.GetConsolePath() + "/NGConsole/Modules/Dummy/DummySettings.asset");
    }

    public override void    OnDisable()
    {
        base.OnDisable();

        this.settings.Uninit();
    }
}
```

The second way merge your data with **NG Settings**.

- Second way:
    - Implement a serializable class with your settings.
    - Add a public field with the class you just created in the partial class *NGSettings*.
    - In your *Module* or your *EditorWindow*, add a field of type *SectionDrawer.*
    - In the method OnEnable(), instantiate the field. It requires 2 arguments, the first is the name of your section and second is the type of your serializable class.
    - In the method OnDisable(), call the method Uninit() from your field of type *SectionDrawer.*

```
using System;
using UnityEngine;

namespace NGToolsEditor
{
    public partial class NGSettings : ScriptableObject
    {
        // This is the class containing your data. Because it is merged into NG
Settings, NG Tools will save it.
        [Serializable]
        public class TestSettings : Settings
        {
            public float    floatSetting = 70F;
            public string   stringSetting = "Test";
            public GUIStyle style;

            protected override void InitGUI()
            {
                this.style = new GUIStyle(GUI.skin.button);
            }
        }
        public TestSettings test = new TestSettings();
    }
}

using NGToolsEditor.NGConsole;
using System;
using UnityEngine;

[Serializable]
[VisibleModule(50)]
public class DummyModule : Module, IStreams
```

```
{
    [NonSerialized]
    private SectionDrawer sectionDrawer;

    public override void    OnEnable(NGConsole editor, int id)
    {
        base.OnEnable(editor, id);

        this.sectionDrawer = new SectionDrawer("Test Section",
typeof(NGSettings.TestSettings));
    }

    public override void    OnDisable()
    {
        base.OnDisable();

        this.editorDrawer.Uninit();
    }
}
```

The first way is a bit complex, but it automatically handles the drawing for you. Use it when you want to create your own settings and put them in a different asset than the one used by **NG Tools**.

The second way is also a bit complex, the main difference with the previous method is your settings are shared in the same asset as **NG Tools**.

See: Implementation in *NGConsole*, *MainModule*, *RemoteModule*.

2.8 NG Network

2.8.1 TCP Listener

Network is handled through a TCP connection, **NG Tools** provides the class *AbstractTcpListener* to implement the behaviour. It only cares about accepting new clients.

The content is managed by the class *Client* which just receives and parses packets.

Every server in **NG Tools** requires a TCP connector, it is done by assigning an instance of an *AbstractTcpListener* in the listener field of the server (Notice that they are both inheriting from MonoBehaviour).

Here is the implementation of *DefaultTcpListener*, the only built-in in **NG Tools**:

```
using System;
using System.Net;
using System.Net.Sockets;

namespace NGTools.Network
{
    public class DefaultTcpListener : AbstractTcpListener
    {
        public override void    StartServer()
        {
            this.tcpListener = new TcpListener(IPAddress.Any, port);
            this.tcpListener.Start(this.backLog);
```

```
            this.tcpListener.BeginAcceptTcpClient(new
AsyncCallback(this.AcceptClient), null);
        }

        private void    AcceptClient(IAsyncResult ar)
        {
            // TcpListener is null when the server stops.
            if (this.tcpListener == null)
                return;

            Client  client = new
Client(this.tcpListener.EndAcceptTcpClient(ar));
            this.clients.Add(client);

            this.tcpListener.BeginAcceptTcpClient(new
AsyncCallback(this.AcceptClient), null);
        }
    }
}
```

As you can see, it only creates a TcpListener from the .Net framework, then wait for clients.

In the case a TcpListener does not fit your needs, you may need to implement your own version of a TCP listener.

Note: **NG Tools** contains a built-in *DefaultTcpListener.*

See: Refer to the class *BaseServer, Packet* and *Client* for more informations.

2.9 NG Hub

2.9.1 HubComponent

The class *HubComponent* displays some GUI in **NG Hub** in order to achieve a task, displaying data, buttons or else.

How to implement a *HubComponent*:

- Create a class inheriting from *HubComponent*.
- Add attribute *Serializable* and Category. The latter is used by **NG Hub** for listing, it sorts the types in a nice way.
- Define the default constructor and call the parent constructor. The parent constructor requires 3 arguments, a string for the name, a boolean defining if your component is editable and a last boolean defining if the editing window must close if it loses the focus.
- [Optional] Override the methods Init or Uninit if you have things to initialize.
- [Optional] Override OnPreviewGUI() if you want to change the default preview GUI. The preview is invoked when editing **NG Hub**.
- [Optional] If your component is editable, override OnEditionGUI() to display GUI that will let you alter your component.
- Override OnGUI() to display things in **NG Hub**.
- [Optional] Define the method "CanDrop" using the following signature: private static bool CanDrop(). This method is called when the user is dragging an *Object*. Therefore you must

override the method InitDrop(), which is invoked when the user drops an *Object* in your component.

An example of a *HubComponent*:

```
using System;
using UnityEngine;

namespace NGToolsEditor.NGHub
{
    [Serializable, Category("Test")]
    public class TestComponent : HubComponent
    {
        [Exportable]
        public string   value;

        public  TestComponent() : base("Test", true, false)
        {
        }

        public override void    Init(NGHubWindow hub)
        {
            base.Init(hub);

            // Init things here
        }

        public override void    Uninit()
        {
            base.Uninit();

            // Uninit things here
        }

        public override void    OnPreviewGUI(Rect r)
        {
            GUI.Label(r, "Test Preview (\"" + this.value + "\")");
        }

        public override void    OnEditionGUI()
        {
            this.value = EditorGUILayout.TextField("Test Value", this.value);
        }

        public override void    OnGUI()
        {
            if (GUILayout.Button("Test Button") == true)
            {
                Debug.Log("Test value " + this.value + "!");
            }
        }

        public override void    InitDrop(NGHubWindow hub)
        {
            base.InitDrop(hub);

            // Do things with the class DragAndDrop.
        }

        private static bool     CanDrop()
        {
```

```
            // Accept drop if the user is dragging a GameObject.
            return DragAndDrop.objectReferences.Length > 0 &&
DragAndDrop.objectReferences[0] is GameObject;
        }
    }
}
```

Note: **NG Hub** contains 6 built-ins *HubComponent*: *TimeScaleComponent*, *MenuCallerComponent*, *LoadSceneComponent*, *AssetShortcutComponent*, *ScenesComponent* and *NavSelectionComponent*. Note that the 2 last were created for **NG Scenes** and **NG Nav Selection**

See: Refer to *LoadSceneComponent* or *AssetShortcutComponent* for real implementation handling drag & drop.

See: Refer to *MenuCallerComponent* to see how to persist **Unity** *Object*.

2.10 Export/Import settings

Exports any portion of your settings through a fine-grained tree. The same way, import any settings you have exported.

To expose your settings to the export wizard, **NG Tools** provides 2 attributes:

- *ExportableAttribute* exposes a field to the export wizard or sets the behaviour of a class when exported.

An example of *ExportableAttribute*:

```
using NGToolsEditor.NGConsole;

public class ExampleModule : Module
{
    [Exportable]
    public int      aInteger;
    [Exportable]
    public string   aString;
}
```

- *HideFromExportAttribute* prevents a class to be exported when the class is in an exportable field array.

An example of *HideFromExportAttribute*:

```
using NGToolsEditor.NGConsole;

public class ExampleModule : Module
{
    public class ClassA
    {
    }

    // This class will be excluded from the list when exported.
    [HideFromExport]
    public class SpecialA : ClassA
    {
    }
```

```
    // An exportable field array.
    [Exportable]
    public ClassA[] array;
}
```

2.11 NG Renamer

2.11.1 TextFilter

Modify a name by implementing a *TextFilter.*

How to implement a *TextFilter*:

- Create a class inheriting from *TextFilter.*
- Define a constructor with 1 argument of type *NGRenamerWindow* and call the parent constructor. The parent constructor requires 3 arguments, a NGRenamerWindow as a dependency, a string as the name, an integer as priority.
- Override Filter().
- Override OnGUI() to display things in **NG Renamer**.
- [Optional] Override Highlight() if you want to highlight some parts of the name.

Here is the implementation of *AddFilter*, a built-in in **NG Renamer**:

```
using UnityEditor;

namespace NGToolsEditor.NGRenamer
{
    public class AddFilter : TextFilter
    {
        public string   text;
        public int      position;
        public string   prefix;
        public string   suffix;

        public  AddFilter(NGRenamerWindow renamer) : base(renamer, "Add", 50)
        {
        }

        public override void    OnGUI()
        {
            using (LabelWidthRestorer.Get(70F))
            {
                EditorGUI.BeginChangeCheck();
                EditorGUILayout.BeginHorizontal();
                {
                    this.text = EditorGUILayout.TextField("Insert", this.text);
                    using (LabelWidthRestorer.Get(25F))
                    {
                        this.position = EditorGUILayout.IntField("At",
this.position);
                    }
                }
                EditorGUILayout.EndHorizontal();

                this.prefix = EditorGUILayout.TextField("Prefix", this.prefix);
```

```
                this.suffix = EditorGUILayout.TextField("Suffix", this.suffix);
                if (EditorGUI.EndChangeCheck() == true)
                {
                    this.enable = true;
                    this.renamer.Invalidate();
                }
            }
        }

        public override string  Filter(string input)
        {
            if (string.IsNullOrEmpty(this.text) == false)
            {
                int pos = this.position;

                if (pos < 0)
                {
                    pos = input.Length + pos + 1;
                    if (pos < 0)
                        pos = 0;
                }
                else if (pos >= input.Length)
                    pos = input.Length;

                input = input.Insert(pos, this.text);
            }

            if (string.IsNullOrEmpty(this.prefix) == false)
                input = this.prefix + input;

            if (string.IsNullOrEmpty(this.suffix) == false)
                input = input + this.suffix;

            return input;
        }
    }
}
```

Note: **NG Renamer** contains 5 built-ins *TextFilter*: *Addfilter*, *ExtensionFilter*, *NumberFilter*, *RemoveFilter* and *ReplaceFilter.*

3 Others

3.1 Guidances

Network

You might encounter issues when using any network system.

If you can not manage to make NG Remote Scene or NG CLI works, you might read the following advices:

- Make sure your device and your computer are in the same network.
- Get the IP of your device and try to ping it from your computer.

- One way to be sure your device is reachable, try to use **Unity's profiler** with your device. For that, make sure that ports 54998 to 55511 are open in the firewall's outbound rules. See https://docs.unity3d.com/Manual/Profiler.html.
- If **Unity's profiler** is reachable, make sure the port you have chosen for *NGServerScene* (17257 by default) or *NGServerCommand* (17254 by default) are also opened.
- **NG Remote Scene** uses the UDP range ports 6547 to 6557 for auto detection.

CPU Spikes

If you perceive CPU spikes in the following cases:

- If you have more than 100k logs. **NG Console** should be able to handle them pretty easily, if not, there might be a problem. Check if you are using a lot of streams with a lot of filters.
- If you have a lot of favorites in **NG Fav** and each time you renamed, create, delete or move a Game Object in Hierarchy.

Contact the author.

3.2 Export/Import settings

You can export or import settings using the fine-grain exporter system in **NG Preferences**. As for now, only **NG Console** and **NG Hub** are exportable.

Note: Before exporting, these windows must be on screen!